RasPi

DESIGN
BUILD
CODE

49

Get hands-on with your Raspberry Pi

TURN YOUR MUSIC INTO HOLOGRAMS

CHECK YOUR MAIL WITH PI

Plus Control Bluetooth with Python

Welcome

Holograms are very cool. Whether it's miniature monsters battling on a circular chessboard or a life-size simulation of an annoying bunk-mate, nothing says Sci-Fi like a moving 3D projection… In this issue, we talk to some rather clever guys from Hacker House who have managed to transform a regular LCD monitor into a 3D 'holographic' music visualiser that's a sure fire head-turner at any party they care to take it to. Swipe left a while to find out just how they achieved it. Other cool tutorials this issue show you how to use SenseHat to create a menu system that interfaces with any Python program you're creating and how to get your Pi talking to different devices using Bluetooth.

Editor

Get inspired
Discover the RasPi community's best projects

Expert advice
Got a question? Get in touch and we'll give you a hand

Easy-to-follow guides
Learn to make and code gadgets with Raspberry Pi

From the makers of Linux User & Developer

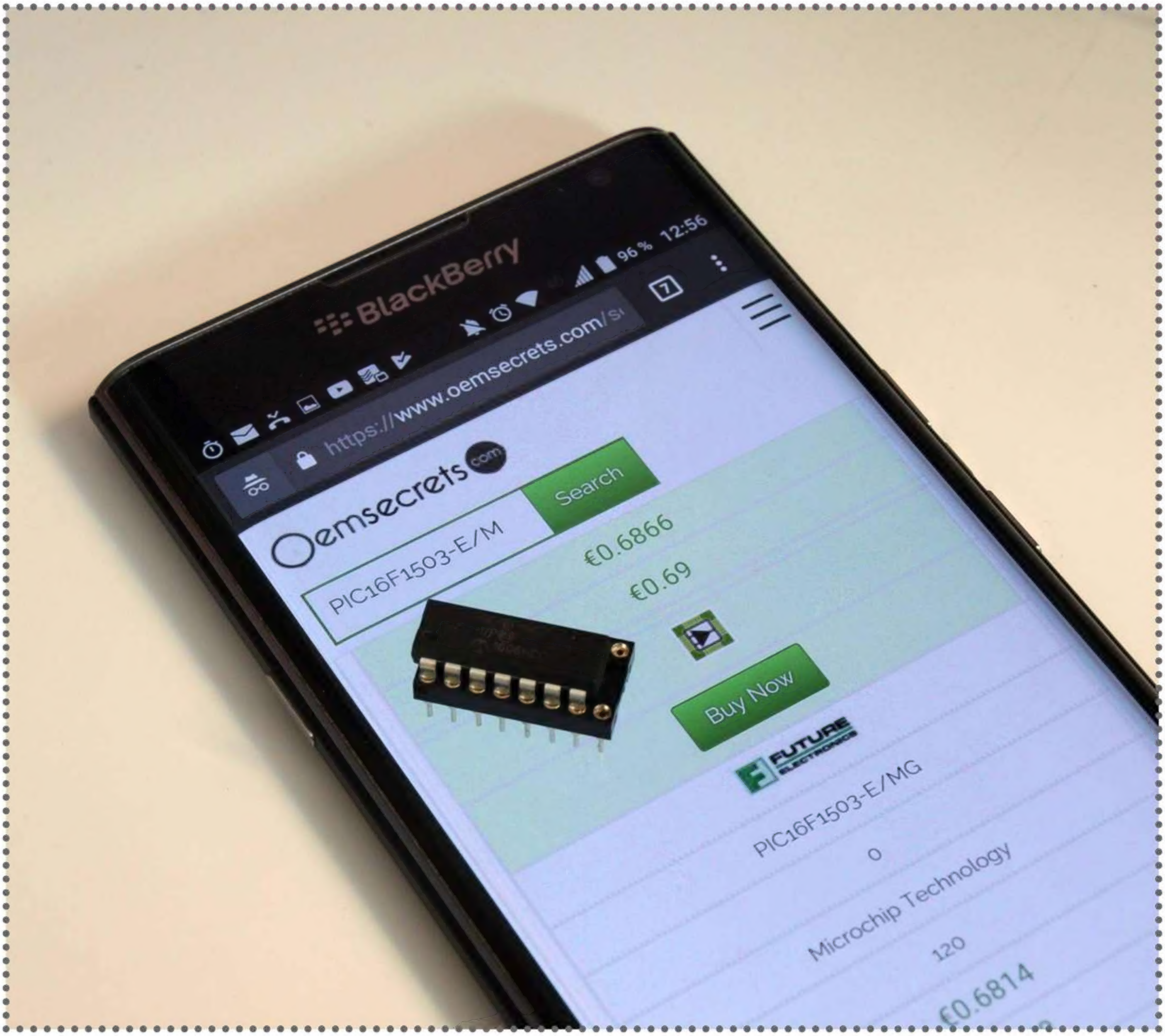
Join the conversation at…

@linuxusermag | Linux User & Developer | linuxuser@futurenet.com

Contents

Raspberry Pi: Send your board to bed in time

Use a low-power, low-cost 8-bit controller to automatically put your Pi to sleep when not in use



Combinatorial process computers became fashionable when the Arduino Yun hit the road; it combined a high-performance ARM SoC with an 8-bit controller intended for handling real-time and low-power tasks. A similar approach can be taken with the Raspberry Pi, combining the power-expensive process computer with a simpler secondary MCU.

The Microchip PIC16F1503 is an extremely convenient candidate: it's backed by an industry-leading IDE (MPLAB X with MCC), it's cheap, can be had in a DIP case and can be programmed with a dirt-cheap programmer.

The following steps outline the reduction of total power consumption. Due to space constraints, we aren't able to discuss the (simple) power regulation circuits.

On the PIC…

Start MPLAB, and create a new PIC project based on the template Microchip Embedded > Standalone Project. Make sure you select the right MCU, and set up the rest of the configuration in a 'sensible' fashion. XC8 is more than good enough as a compiler.

Select the project in the Projects tab, and click the MCC symbol in the toolbar. This opens Microchip's code generator, which greatly simplifies the generation of useful project skeletons. Make sure to select a very high clock rate (I usually go for 16MHz) – the default 500KHz isn't enough to power the I2C protocol.

Next, select the MSSP unit in the Device Resources tab and add it to the project resources in order to start configuring the hardware modules used in the solution. The correct mode is called I2C slave, with the slave address 0x8 being perfectly suited to our needs.

Finally, use the Pin Manager to declare RC3 as

THE PROJECT ESSENTIALS

- PIC 16F1503
- PIC programmer of choice
- Breadboard, wiring

output – its name can be modified in the Pin Module. When done, save the file MyConfig.mc3 in the place recommended by MPLAB. Then click generate to allow MCC to modify the structure of the underlying project. Disable MCC by switching to the Projects tab, and open the file MCC Generated Files > 2c.c.

Microchip's code generator spawns a virtual I2C EEPROM, overloading the PIC 16F1503's memory. Thus, this code must go. Removing I2C_StatusCallback is especially important: the harness code generated by MCC will invoke it whenever there is data that needs to be processed.

This first part of the code handles reading. Writing can be handled by placing the byte to be sent to the master into the SSP1BUF register, like so:

```
case I2C_SLAVE_READ_REQUEST:
  SSP1BUF=lastWakeupState;
  break;
case I2C_SLAVE_READ_COMPLETED:
  break;
```

The exact variables and constants used in the example are not important; just make sure they exist in accordance to the sample code in this tutorial.

With that out of the way, the next destination is the main loop. When programmed in C, microcontrollers generally behave like basic command line programs – after start-up, main is run until Godot smites the unit:

```
void main(void) {
    SYSTEM_Initialize();
    INTERRUPT_GlobalInterruptEnable();
    INTERRUPT_PeripheralInterruptEnable();
```

Microchip lets developers gate 'un-needed' parts of the MCU. Due to this, we need to start out by enabling various elements. After that, the RC3 pin is raised to signal that the MCU must be powered up:

```
    currentState=S_RUNNING;
    lastWakeupState=I2CARAISE;
    IO_RC3_SetHigh();
```

The next bit of code is the actual main loop. Its execution rate is limited via a call to the Microchip-specific macro __delay_ms. One run of the loop takes 100ms and whatever time is needed for processing. The rest of the waiting loop is super-simple. We check if we are asleep: if that is the case, the remaining sleep-time timer is decremented. Finally, the state of the power pin is adjusted accordingly:

```
    while(1==1) {
        if(currentState==S_GOINGTOSLEEP) {
            timerGotoSleep--;
            if(timerGotoSleep==0) {
                currentState=S_SLEEPING;
                    IO_RC3_SetLow();
            }
        }
```

Shutting down the process computer is not an instantaneous process – our program accommodates for this by powering off the Raspberry Pi only after the specified delay has passed.

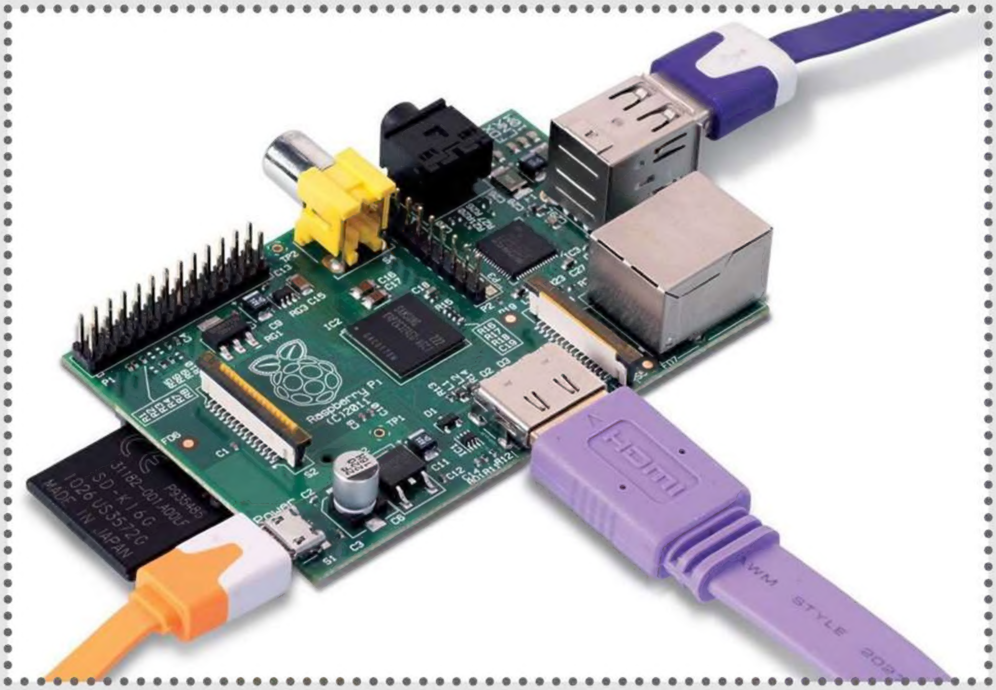
Build a testing rig

When working with PIC microcontrollers, building a testing rig that exposes the various GPIO pins and an interface to the programmer is a sure-fire way to save yourself a lot of stress. These testing rigs don't need to win any engineering prizes – you're fine as long as it's more robust than a breadboard circuit.

```
        else if(currentState==S_SLEEPING)
{
                timerSleep--;
                if(timerSleep==0) {
                    currentState=S_RUNNING;
                    lastWakeupState=I2CAALARM;
                        IO_RC3_SetHigh();
                }
            }
            __delay_ms(100);
        }
        return;
```

Wire it up

With that, only a small engineering effort is needed. Grab yourself a relay and control its activator from the RC3 pin. Next, connect the SCL and SDA lines of the PIC to the same ones on the process computer, and ensure that the two power supplies are independent from one another.



From that moment onward, your PIC can power itself off to be woken up again later. For testing, a more convenient approach can be connecting an LED to the output and monitoring the voltage levels with an oscilloscope in ROLL mode.

First, use the raspi-config tool to enable the I2C interface in Interfacing Options. After the obligatory reboot, connect the PIC to the relevant signalling lines. Our chip can live in the 3V3 domain. Next, use the i2cdetect command to detect the presence of the chip. Its message about wreaking total havoc on the bus can be ignored.

Figure 1

```
char writeCounter;
void I2C_StatusCallback(I2C_SLAVE_DRIVER_
  STATUS i2c_bus_state) {
    switch (i2c_bus_state) {
        case I2C_SLAVE_WRITE_REQUEST:
            writeCounter=0;
            timerSleep=0;
            break;

        case I2C_SLAVE_WRITE_COMPLETED:
                timerSleep=
  (timerSleep<<8)+I2C_slaveWriteData;
                if(writeCounter==1) {
                    timerGotoSleep=300;
                    currentState=S_
  GOINGTOSLEEP;
                }
            writeCounter++;
            break;
```

LEFT The write routine is long, but not complex

With that out of the way, the next test involves wiring up a small demonstration program. UNIX treats the I2C bus like any other device, and addresses it via a device in the /dev tree. Due to that, a group of constants is needed along with a file descriptor:

```
#define I2C_DEVICE "/dev/i2c-1"
#define I2C_PWC_ADDR 0x08
#define ERROR -1
static int i2c_fd_;
```

Next up is a small method which takes care of wiring up the UNIX-level plumbing:

```
void init_hardware() {
  wiringPiSetup();
  i2c_fd_ = wiringPiI2CSetupInterface (I2C_
DEVICE, I2C_PWC_ADDR);
  if (i2c_fd_ == ERROR) {
    printf ("setup fail!\n");
```

main() contains the actual core of our program. Keep in mind that I2C is a register-orientated protocol, which we implemented only partially in order to save space on the MCU. First, a random register is read in order to trigger the PIC to respond with its wakeup reason:

```
void main() {
  init_hardware();
  wiringPiI2CWriteReg8(i2c_fd_, 0x00, 0x00);
  int s = wiringPiI2CRead (i2c_fd_);
  printf ("Wakeup reg: %i\n", s);
```

After that, we write out a two-byte value describing the

sleep time in 100msec ticks. The address register is used as the first of the two values in order to simplify read handling on the PIC:

```
wiringPiI2CWriteReg8(i2c_fd_, 0, 255);
```

Due to space constraints, the program demonstrated here is but a bare minimum viable implementation. In practical code, you could analyse one or more of the additional GPIO pins in order to reboot the process computer if, for example, a power button is pressed or an attenuator is toggled. An additional benefit of this design is that the PIC can be expanded programmatically; one customer of mine uses the PIC to interact with various temperature sensors.

Holographic Audio Visualiser

Make your music come alive with this futuristic holographic-style visualiser and track changer

With some acrylic sheets mounted on a wooden frame, the hacker duo at Hacker House have managed to transform a regular LCD monitor into a 3D 'holographic' music visualiser that's a sure head-turner at any party. The device projects an image from a monitor down onto an acrylic pyramid, or frustum, which then creates the holographic parallax effect. To add to the futuristic aura of the device, Aaron and Davis have connected a Flick board from Pi Supply, which enables them to use Jedi-like hand gestures to control playback, volume and visualisations. The Flick board is connected to a Raspberry Pi 3 that processes the gestures and sends them to the visualiser, which is running on a more powerful computer that can smoothly run WebGL graphics at a decent frame rate. The enterprising pair have done an excellent job of documenting the entire build with a detailed construction video and a thorough illustrated guide (Hackster.io, **http://bit.ly/HoloVisualiser**). At first glance, the physical construction of the project might seem quite daunting, since creating the frame requires several different lengths of wood and the acrylic sheets need to be scored to exact dimensions dependent on the size of the monitor projecting the visualisation. The construction part of the project also requires a slew of 3D-printed parts to hold the frame, the monitor and the Flick board, along with the Raspberry Pi. However, the duo has shared the .STL files for the custom parts along with the code.

In fact, their instructions are so easy to follow that a group of school pupils have been able to

Aaron Tainter & Davis Lau is a university student and an experienced modder with a passion for console hacks. He does not claim to have perfect mods, but he is a mod purist and strives to keep his mods looking as original as he possibly can

successfully replicate the project and build their own holographic visualiser (Instructables, http://bit.ly/HoloProjector) as part of a school project. Because the visualiser is essentially controlled by web requests, the students have even managed to change some of the Pi code to get the device to work with buttons instead of the Flick gesture-control pad.

What was the inspiration behind the project?

We acquired a gesture-control board and we were brainstorming things to use it for. I had done some 3D audio-visualiser graphics with WebGL in the past for a personal project. At the same time, we saw some videos online about how to make mini hologram-projectors with a CD case and a phone. We thought it would be interesting to merge those things together to make a really futuristic visualiser.

THE PROJECT ESSENTIALS

Raspberry Pi 2

Fridge

Raspberry Pi 3

Pi Supply Flick gesture breakout

24-inch computer monitor

0.093-inch acrylic sheet n 0.118-inch black foam PVC sheet

Various sizes of wooden planks

Handful of screws

Hand drill

Spray paint

Some 3D-printed brackets

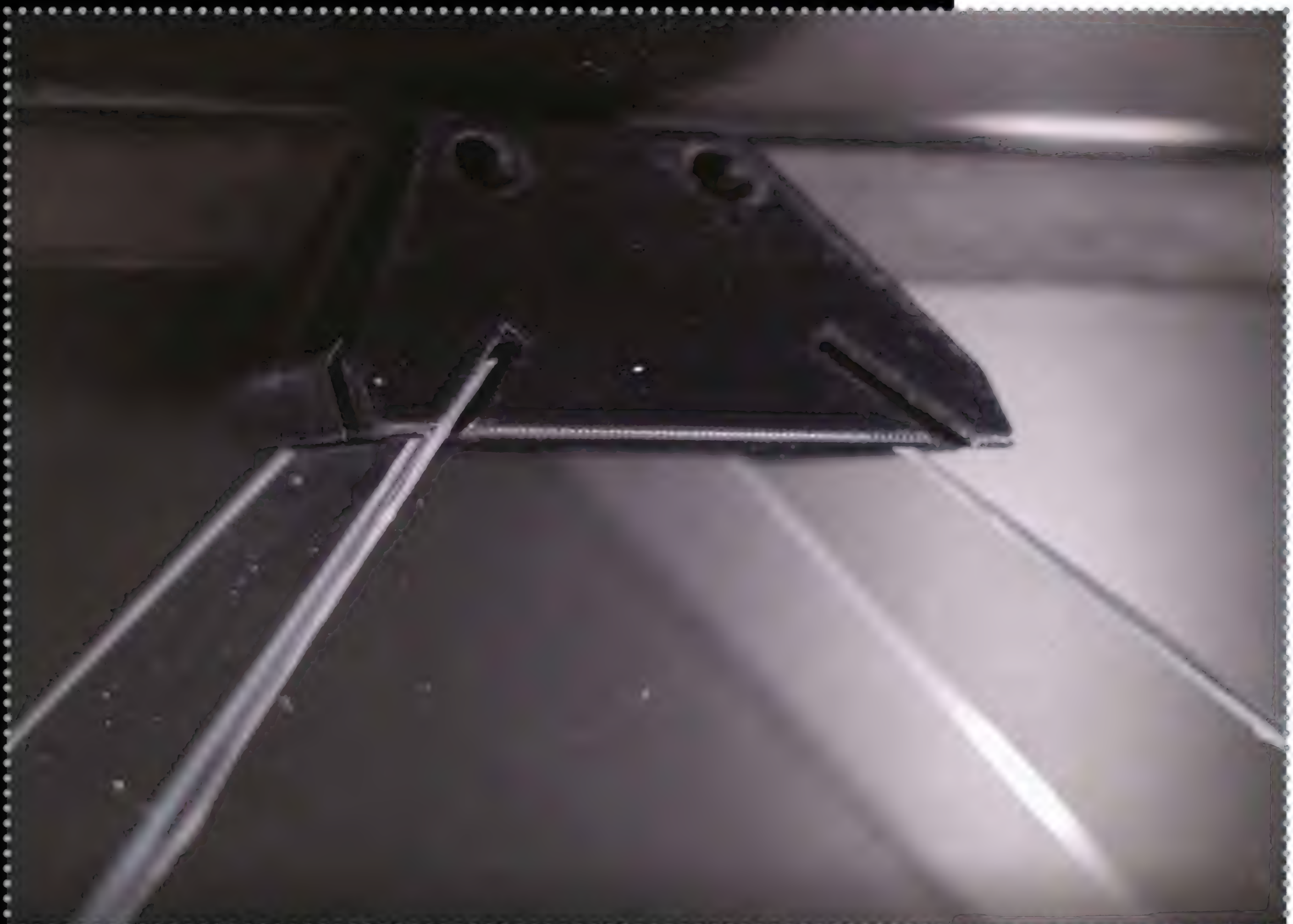
What was the most challenging part of the project? Is there anything you would have done differently?

The construction was pretty straightforward, but the code took quite a bit of time to write. I would have liked to add more animations or a simple interface for anyone at home to add their own visualiser.

Any particular reason for using a computer to run the visualiser instead of the Pi?

We initially tried running the WebGL graphics on the Pi, but it wasn't able to run them at a smooth frame rate. Ideally, we'd power everything from one device, but the control board didn't have an SDK that could be run on a normal UNIX machine.

Below The creation of the frustum was one of the trickiest parts of the build. A 3D-printed holder, bolted to the centre of the frame, was used to slide in the three acrylic sheets at the proper angles for the projection effect to work.



Do you have any plans to extend the project?

I was originally going to use the Spotify API for music, but I discovered that it only streams 30-second sound clips. We had to use the Soundcloud API because it was the only thing that I could find that can stream full songs, but it's mostly independent stuff. In the future, I might try to stream audio from local files or a Bluetooth connection with my phone, so that I can play more popular music.

This project involves a lot of physical construction. What things should people keep in mind before trying to build it?

The Plexiglass is very difficult to cut, so I would make sure to use a thin piece and create some paper templates to verify that everything fits properly. If they

Below The base and the back side are covered with black PVC sheets that block light and also help the 'holographic' effect stand out more strongly in front of the dark background. The sheets have small notches to make them fit between the 3D-printed brackets.

don't fit perfectly in place, you can always use some clear tape at the bottom to hold them together.

What kind of projects interest you? What does the Hacker House have for us in the pipeline?

I'm really interested in the practical applications of AI and computer vision. I'd like to make more smart-home appliances with AI, either a custom AI or Alexa/ Google Home integration.

In the past, we also wrote code for an autonomous cooler, but it navigated based on GPS waypoints. We want to extend that code with some computer vision to give a few other objects some basic autonomous vehicle capabilities.

Further reading

They run a Youtube channel (**www.youtube.com/c/HackerShack**) and a website called Hacker Shack (www.thehackershack.com) where they share interesting DIY builds.

Sense HAT: Create a physical menu system

Take your Python code to the next level with some physical computing and the Raspberry Pi Sense HAT

The Sense HAT is an official add-on board for the Raspberry Pi, offering a whole host of built-in sensors including a gyroscope, accelerometer and magnetometer, plus temperature, barometric pressure and humidity sensors. What we're most interested in, though, is the fancy 8x8 RGB LED matrix and five-button joystick on top. Using an array of 8x8 images we're going to create a menu system that is operable with the five-button joystick and select button.

We'll start off with a simple four-option menu system where up, down, left and right analogue positions enable the user to choose from four different menu screens made up of 8x8 pixel art. Pushing in the analogue stick button will select the desired option. We'll then take a look at building on this to create potentially unlimited menu options.

This menu system can then interface with any Python program you're creating, providing a physical user interface which we think you'll agree proves to be a much more appealing user experience than entering a number in a regular Python shell window command prompt.

01 Install the HAT

When attaching the HAT to your Pi be sure to screw the standoffs in place before plugging the HAT onto the Raspberry Pi's GPIO pins; this will avoid it bending on one side and potentially breaking the pins. If for some reason you need to remove the Sense HAT at some point, be very careful as the pin header tends to come off with it. Once attached, we'll need to install the Sense HAT Python module with:

THE PROJECT ESSENTIALS

Raspberry Pi

Sense HAT (www.raspberrypi.org/products/sense-hat)

Python 3

Sense HAT for Python 3 (https://pypi.org/project/sense-hat)


```
sudo apt-get install sense-hat
```

02 Set up modules

We may want to use the sleep command for added effect when loading pixels onto the LED matrix, so we'll need to import the time module. We're also going to use a bunch of functions from the sense-hat module that we installed previously. These are mostly for recognising button-presses on the analogue stick:

```
from time import sleep
from sense_hat import SenseHat, ACTION_
  PRESSED, ACTION_HELD, ACTION_RELEASED
```

This enables us to monitor whether a button is pressed (down), held or released.

03 Set up the variables

We'll need a new instance of SenseHat, which we'll call sense for simplicity. We'll also need to set up variables for each colour of pixel we want to draw on the LED matrix; we've gone with red, green and blue here, but you can use any RGB colour values:

```
sense = SenseHat()
```

Displaying alpha-numeric digits on the Sense HAT

To display a single-character on the LED matrix use sense.show_letter("C") where C is any number or letter that takes your fancy. This is far simpler than creating an 8x8 frame of pixels per menu – ideal for creating a quick concept menu.

```
r=(255,0,0)
g=(0,255,0)
b=(0,0,255)
```

04 Menu images - pixel art

Creating images for your menu system based on pixel art can be a challenge in itself. Try looking up some ASCII art for inspiration. Being limited to an 8x8 grid can be a blessing and curse. You'll need a different 'frame' for each menu option, four to begin with.

```
frame1 = [g,g,g,g,g,g,g,g,g,b,b,g,g,b,b,g,g,b,b,g
 ,g,b,b,g,g,g,g,b,b,g,g,g,g,g,g,b,b,g,g,g,g,b,b,b,
 b,b,b,g,g,b,b,b,b,b,b,g,g,g,g,g,g,g,g,g,]
```

Be experimental with your frames. Rather than loading the whole thing at once, the following tip will demonstrate how to slowly load a frame line by line.

05 Animating a frame

Create a variable i=0 and add the following:

```
for i in range(8):
    sense.set_pixel(0,i,g)
    sense.set_pixel(1,i,g)
    sense.set_pixel(7,i,g)
    sleep(0.2)
    i=i+1
```

Filling in the gaps above for rows 2-6 will fill the LED matrix one row at a time, incrementally.

06 Showing off with animations

Another neat trick is the flashing frames. By using the

below technique and adding more frames you can create a flick-book style animation that looks quite impressive:

```
sense.clear()
while True:
    sense.set_pixels(frame1)
    sleep(0.5)
    sense.set_pixels(frame2)
    sleep(0.5)
```

```
sense.clear()
i=0
#fill column one at a time
for i in range(8):
    sense.set_pixel(0,i,g)
    sense.set_pixel(1,i,g)
    sense.set_pixel(2,i,g)
    sense.set_pixel(3,i,g)
    sense.set_pixel(4,i,g)
    sense.set_pixel(5,i,g)
    sense.set_pixel(6,i,g)
    sense.set_pixel(7,i,g)
    sleep(0.2)
    i=i+1
```

07 Set up buttons

Our buttons will be set up in three sections of code. We'll have a function for displaying the 8x8 pixel image so that we can look at different menu options before selecting one. We'll then have a separate function for actioning a button press; when pushed/released we'll call up a specific action depending on which image is currently highlighted.

An easy way to experiment with this code without connecting your Sense HAT to the Pi is by running it on a web emulator. Trinket.io has a fantasic emulator that makes the trial and error process far quicker, because

you can run the code on your favourite Linux distro. See https://trinket.io/sense-hat.

08 Create menu highlighting

When a menu function is called (upMenu, rightMenu, leftMenu or downMenu), we're accepting a single parameter (event) which will let us know if the button has been released. We're using 'released' rather than 'pressed' simply because it feels more natural.

If the button (in this case, 'up') is released we're printing a message to the console, mainly for debug purposes so that we know the code is working, then we're clearing the LED matrix and displaying a new pixel image from frame1. If the middle button is pressed we call our second function, upCreate; again, we'll have a create functions for down, left and right as well.

09 Create menu highlighting – the code

We'll have four instances of the following code, one for each menu option (up/down/left/right). Change the function names accordingly and insert the correct

File Edit Format Run Options Window Help

```
def downMenu(event):
  sense.clear()
  if event.action == ACTION_RELEASED:
    sense.set_pixels(snowflake)
    print("*Down option highlighted.*")
    #menu item 2
    sense.stick.direction_middle = downCreate

def downCreate(event):
  if event.action == ACTION_RELEASED:
    print("*Down option selected.*")
    #Action code goes here

def rightMenu(event):
  if event.action == ACTION_RELEASED:
    sense.clear()
    sense.set_pixels(diamond)
    print("*Right option highlighted.*")
    sense.stick.direction_middle = rightCreate

def rightCreate(event):
  if event.action == ACTION_RELEASED:
    print("*Right option selected.*")
```

variable for the pixel art:

```
def upMenu(event):
  if event.action == ACTION_RELEASED:
    print("*Up option highlighted.*")
    sense.clear()
    sense.set_pixels(frame1)
    sense.stick.direction_middle = upCreate
```

10 Menu selection code

Very similar to the previous function, we have one parameter for a button release, we're printing to console for debugging, and then there's a space to insert whatever actionable code you require. You could essentially put a while Python statement after that comment if you wanted to:

```
def upCreate(event):
  if event.action == ACTION_RELEASED:
    print("*Up option selected.*")
    #Action code goes here
```

11 Operating the buttons

Now that we've got our buttons set up to display menus and action code, we'll need to provide Python with a way of knowing if they've been pressed in the first place. Using our sense object we can listen for events stick_direction_up, stick_direction_down, stick_direction_left and stick_direction_right:

```
sense.stick.direction_up = upMenu
sense.stick.direction_down = downMenu
```

Shake to clear

Introduce the Yaw axis to know when the Pi travels up or down vertically by reading the z acceleration with the sense.get_accelerometer_raw function. This is a nice way of introducing a 'shake to clear' function, which can reset the menu screen. Introduce z = acceleration['z'] to the while loop in our penultimate tutorial step, and round the values with z=round(z, 0). Then add an if statement along the lines of if z == 1: clearMenu, provided clearMenu was a function set to clear the screen with sense.clear().

```
sense.stick.direction_right = rightMenu
sense.stick.direction_left = leftMenu
```

12 Keeping it running

In order for the above to work and to keep our programming running we'll use a loop:

```
while True:
    pass
```

Or to be more efficient and save CPU resources use:

```
while True:
  time.sleep(1)
```

menu-companion-advanced.py - /h...nu-companion-advanced.py (3.5.3)

File Edit Format Run Options Window Help

```
def optionMenu(event):
  sense.clear()
  if event.action == ACTION_RELEASED:
    #sense.set_pixels(frame1)
    #menu item 1
    sense.stick.direction_right = optionMenu2
    sense.stick.direction_left = optionMenu4

def optionMenu2(event):
    if event.action == ACTION_RELEASED:
        sense.set_pixels(frame2)
        #menu item 2
        sense.stick.direction_right = optionMenu3
        sense.stick.direction_left = optionMenu

def optionMenu3(event):
    if event.action == ACTION_RELEASED:
        sense.set_pixels(frame2)
        #menu item 2
        sense.stick.direction_right = optionMenu4
        sense.stick.direction_left = optionMenu2

def optionMenu4(event):
```

13 Advanced menu system

To improve this system we may want to increase the number of menu options available. Rather than using up, down, left and right to highlight menu options we could stick with left and right for back and forth by introducing scrolling. With this method our options are

potentially unlimited.

We could do this by adding alternate functions to the left and right directional menus, that is by adding sense.stick.direction_right = optionMenu3 to our rightMenu function. We could add as many optionMenu functions as we need, progressing with the right stick and not forgetting to add sense.stick.direction_left optionMenu2 to give users an option to go back using the left stick.

14 Call independent Python programs

We can implement any code into the actionable functions and it'll occur whenever we select that menu option. However, you might instead want to launch a completely independent Python program when certain menu options are selected.

Python's included os module takes care of this; if we add import os to the top of our code we can call upon independent Python programs in each menu option with the following:

```
os.system('testy.py')
```

where test.py is the name of the program.

15 Building from here

We have the option of tilting the Pi left and right to change menu options using the Sense HAT's accelerometer. The two axes we need to observe are Pitch, which is left/right, and Roll which is up/down, in-line with the analogue stick.

```
while True:
  acceleration = sense.get_accelerometer_
  raw()
```

```
    x = acceleration['x']
    y = acceleration['y']
```

16 Shaking/rolling

We'll need to round off the accelerometer values into floats to make them useable; x=round(x, 0) and y=round(y, 0) will do the trick. We can now use an if statement for x or y being equal to 1 or -1 for left or right.

```
  if x  == 1:
  elif x == -1:
```

Implement a call to an optionMenu function in these if/elif statements and we have a functioning tilt menu.

Check your mail

We show you how to create a mail-checker that can display your current unread emails

Since the Raspberry Pi is such a small computer, it gets used in a lot of projects where you want to monitor a source of data. One such monitor you might want to create is a mail-checker that can display your current unread emails. This issue, we'll look at how to use Python to create your own mail-checking monitor to run on your Pi. We'll focus on the communications between the Pi and the mail server and not worry too much about how it might be displayed. That will be left as a further exercise.

To start with, most email servers use one of two different communication protocols. The older, simpler one was called POP (Post Office Protocol), and the newer one is called IMAP (Internet Message Access Protocol). We will cover both protocols to cover all of the situations that you might run into. We'll start with the older POP communications protocol. Luckily, there is support for this protocol as part of the standard library. In order to start using this protocol, you will need to import the poplib module, and then create a new POP3 object. For example, the following will create a connection to the POP server available through Gmail.

Check your mail

We show you how to create a mail-checker that can display your current unread emails

Since the Raspberry Pi is such a small computer, it gets used in a lot of projects where you want to monitor a source of data. One such monitor you might want to create is a mail-checker that can display your current unread emails. This issue, we'll look at how to use Python to create your own mail-checking monitor to run on your Pi. We'll focus on the communications between the Pi and the mail server and not worry too much about how it might be displayed. That will be left as a further exercise.

To start with, most email servers use one of two different communication protocols. The older, simpler one was called POP (Post Office Protocol), and the newer one is called IMAP (Internet Message Access Protocol). We will cover both protocols to cover all of the situations that you might run into. We'll start with the older POP communications protocol. Luckily, there is support for this protocol as part of the standard library. In order to start using this protocol, you will need to import the poplib module, and then create a new POP3 object. For example, the following will create a connection to the POP server available through Gmail.

need a quick summary of what is on the open server you can execute the stat() method:

```
my_pop.stat()
```

This method returns a tuple consisting of the message count and the mailbox size. You can get an explicit list of messages with the list() method. You have two options for looking at the actual contents of these emails, depending on whether you want to leave the messages untouched or not. If you want to simply look at the first chunk of the messages, you can use the top() method. The following code will grab the headers and the first five lines of the first message in the list.

```
email_top = my_pop.top(1, 5)
```

This method will return a tuple consisting of the response text from the email server, a list of the headers and the number of requested lines, and the octet count for the message. The one problem with the top() method is that it is not always well implemented on every email server. In those cases, you can use the retr() method. It will return the entire requested message in the same form as that returned from top().

Once you have your message contents, you need to decide what you actually want to display. As an example, you might want to simply print out the subject lines for each message. You could do that with the following code.

```
for line in email_top[1]:
if 'Subject' in i:
```

```
    print(i)
```

You need to explicitly do the search because the number of lines included in the headers varies between each message. Once you're done, don't forget to execute the quit() method to close down your connection to the email server. One last thing to consider is how long the email server will keep the connection alive. While running test code for this article, it would frequently time out. If you need to, you can use the noop() method as a keep-alive for the connection.

As mentioned previously, the second, newer, protocol for talking to email servers is IMAP. Luckily, there is a module included in the standard library that you can use, similar to the poplib module we looked at above, called imaplib. Also, as above, it contains two main classes to encapsulate the connection details. If you need an SSL connection, you can use IMAP4_SSL. Otherwise, you can use IMAP4 for unencrypted connections. Using Gmail as an example, you can create an SSL connection with the following code.

"Don't forget to execute the quit() method to close down your connection to the email server"

```
import imaplib
import getpass
my_imap = imaplib.IMAP4_SSL('imap.gmail.com')
```

As opposed to poplib, imaplib has a single method to handle authentication. You can use the getpass module to ask for the password.

```
my_imap.login('my_username@gmail.com',
```

```
getpass.getpass())
```

IMAP contains the concept of a tree of mailboxes where all of your emails are organised. Before you can start to look at the emails, you need to select which mailbox you want to work with. If you don't give a mailbox name, the default is the inbox.

This is fine since we only want to display the newest emails which have come in. Most of the interaction methods return a tuple that contains a status flag (either 'OK' or 'NO') and a list containing the actual data. The first thing we need to do after selecting the inbox is to search for all of the messages available, as in the following example.

```
my_imap.select()
typ, email_list = my_imap.search(None, 'ALL')
```

The email_list variable contains a list of binary strings that you can use to fetch individual messages. You should check the value stored in the variable typ to be sure that it contains 'OK'. To loop through the list and select a given email, you can use the following code:

```
for num in email_list[0].split():
typ, email_raw = my_imap.fetch(num,
  '(RFC822)')
```

The variable email_raw contains the entire email body as a single escaped string. While you could parse it to pull out the pieces that you want to display in your email monitor, that kind of defeats the power of Python.

Again, available in the standard library is a module called email that can handle all of those parsing issues.

You will need to import the module in order to use it, as in the example here.

```
import email
email_mesg = email.message_from_bytes(email_
  raw[0][1])
```

All of the sections of your email are now broken down into sections that you can pull out much more easily. Again, to pull out the subject line for a quick display, you can use the code:

```
subject_line = email_mesg.get('Subject')
```

There are many different potential items that you could select out. To get the full list of available header items, you can use the keys method, as shown below:

```
email_mesg.keys()
```

Many times, the emails you get will come as multi-part messages. In these cases, you will need to use the get_payload() method to extract any attached parts. It will come back as a list of further email objects. You then need to use the get_payload() method on those returned email objects to get the main body. The code might look like:

```
payload1 = email_mesg.get_payload()[0]
body1 = payload1.get_payload(
```

As with POP email connections, you may need to do something to keep the connection from timing out. If

you do, you can use the noop() method of the IMAP connection object. This method acts as a keep-alive function.

When you are all done, you need to be sure to clean up after yourself before shutting down. The correct way to do this is to close the mailbox you have been using first, and then log out from the server. An example is given here:

```
my_imap.logout()
my_imap.close()
```

You now should have enough information to be able to connect to an email server, get a list of messages, and then pull out the sections that you might want to display as part of your email monitor. For example, if you are displaying the information on an LCD, you might just want to have the subject lines scrolling past. If you are using a larger screen display, you might want to grab a section of the body, or the date and time, to include as part of the information.

What about sending emails?

In the main body of the article, we have only looked at how to connect to an email server and how to read from it. But what if you need to be able to also send emails off using some code? Similar to poplib and imaplib, the Python standard library includes a module called smtplib. Again, similar to poplib and imaplib, you need to create an SMTP object for the connection, and then log in to the server. If you are using the GMail SMTP server, you could use the code

```
import smtplib
import getpass
my_smtp = smtplib.SMTP_SSL('smtp.gmail.com')
```

"When you are all done, you need to be sure to clean up after yourself before shutting down"

```
my_smtp.login('my_email@gmail.com', getpass.
    getpass())
```

This code asks the end user for their password, but if you aren't concerned about security, you could have it hard-coded into the code. Also, you only need to use the login() method for those servers that require it. If you are running your own SMTP server, you may have it set up to accept unauthenticated connections. Once you are connected and authenticated, you can now send emails out. The main method to do this is called sendmail(). As an example, the following code sends a 'Hello World' email to a couple of people.

```
my_smtp.sendmail('my_email@gmail.com',
    ['friend1@email.com', 'friend2@email.com'],
    'This email\r\nsays\r\nHello World')
```

The first parameter is the 'from' email address. The second parameter is a list of 'to' email addresses. If you have only a single 'to' address, you can put it as a single string rather than a list. The last parameter is a string containing the body of the email you are trying to send. One thing to be aware of is that you will only get an exception if the email can't be sent to any of the 'to' email addresses specified. As long as the message can be sent to at least one of the given addresses, it will return as completed. Once you have finished sending your emails, you can clean up with the code:

```
my_smtp.quit()
```

This cleans everything up and shuts down all active connections. So now your project can reply to incoming emails, too.

A guide to controlling Bluetooth with Python

We show you how to get your Raspberry Pi to talk to some other devices – without the wires

One of the great uses of the Raspberry Pi is as a controller for other devices. The Raspberry Pi includes the GPIO bus, which most people use for this purpose. But sometimes, you may need to talk to devices that can't be connected to your Pi by wires. If you have a Wi-Fi hub, you could communicate over it with your devices. This may not be possible in every case: your device may not be able to speak over Wi-Fi, or the device may be low-power and not able to run something as electrically expensive as a Wi-Fi radio. In these cases, another option is to communicate over Bluetooth.

As of writing, there are several Raspberry Pi variants with Bluetooth integrated into the system. Any of the Raspberry Pi 3 versions, or the Raspberry Pi Zero W, have built-in Bluetooth radios. With the earlier variants, you would need to use a USB dongle to give your Raspberry Pi Bluetooth capabilities – we won't be covering this option in too much detail in this article. In terms of software, everything you need should be

available within the Raspbian package repository.

Install bluetooth

The first step is to ensure you have all of the required packages installed. You'll want to make sure your entire system is updated first, with the following commands:

```
sudo apt-get update
sudo apt-get upgrade -y
sudo apt-get dist-upgrade -y
sudo rpi-update
```

Once your Raspberry Pi is fully updated, you can install everything needed with one meta-package:

```
sudo apt-get install pi-bluetooth
```

Not only do you get the required drivers for the Bluetooth stack, but you also get several utilities that you can use to verify your setup.

Explore controllers

With everything installed, you can use one of the included command-line utilities to check that it's all working correctly. The main utility for this purpose is bluetoothctl. When you run it, you get dropped into an interpreter where you can issue commands to the Bluetooth stack. From here, you can use the list command to see what Bluetooth controllers are on your Raspberry Pi. If you do have multiple dongles plugged in, you can use the select command to pick which one should be used.

You'll want to make sure that the controller is

Why Python?

It's the official language of the Raspberry Pi. Read the docs at https://www.python.org/doc/

powered up and active before starting a scan of what devices are available locally. You can do this with the following simple commands.

```
power on
Changing power on succeeded
agent on
Agent registered
scan on
Discovery started
....
```

You should now see a stream of devices as they become visible to the Raspberry Pi. This will include device IDs, which can be used to connect to them. This is handled with the connect command, followed by the hexadecimal ID for the required device.

Use a GUI application

Along with command line tools like bluetoothctl, there are GUI applications that you can use to manage Bluetooth devices and the way your Raspberry Pi connects to them. If you're using a Raspberry Pi with a display as the front-end for your project, these can be very useful to make sure everything is set up correctly. One popular GUI application is blueman. With this utility, you can scan to see what devices are available in your vicinity, and make connections to those that you are interested in using. You can pull up more detailed information on these devices, as well as copying files to and from them.

These are important tools that you should use to debug connections before diving into using your own

code to manage any communication between the Raspberry Pi and the Bluetooth device. You don't want to have to debug your code without establishing that the underlying Bluetooth connection does work, at least in principle. Otherwise, you may end up wasting time trying to fix non-existent bugs in your code when there is some other hardware issue happening.

Intro to Python

While all of these tools are really useful, the real purpose of this tutorial is to give you the ability to add Bluetooth functionality to your own Python code. This way, you can connect multiple devices together and have them managed from your Raspberry Pi, which we're sure you'll find useful. The first step is to make sure that you have the Bluetooth module for Python installed, with the following command:

```
sudo apt-get install python-bluetooth
```

Once it's installed, you can import it into your code with:

```
import bluetooth
```

Since Bluetooth is a type of network stack, a lot of the language around Bluetooth coding should seem very familiar to anyone who has written networking code before. You need to create socket objects and use them to open connections to other devices. Then you can send and receive data across this socket. So the first step is to create the socket:

```
server_socket = bluetooth.
  BluetoothSocket(bluetooth.RFCOMM)
```

"While all of these tools are really useful, the real purpose of this article is to give you the ability to add Bluetooth functionality to your own codex"

Once the socket is created, you can listen for incoming connections, like this:

```
port = 1
server_socket.bind((“’,port))
server_socket.listen(1)
```

The above code binds the socket to a selected port, and then listens on that port for incoming connections. After that, you can accept incoming connections. As you can see below, you get a client socket and an address from this incoming connection:

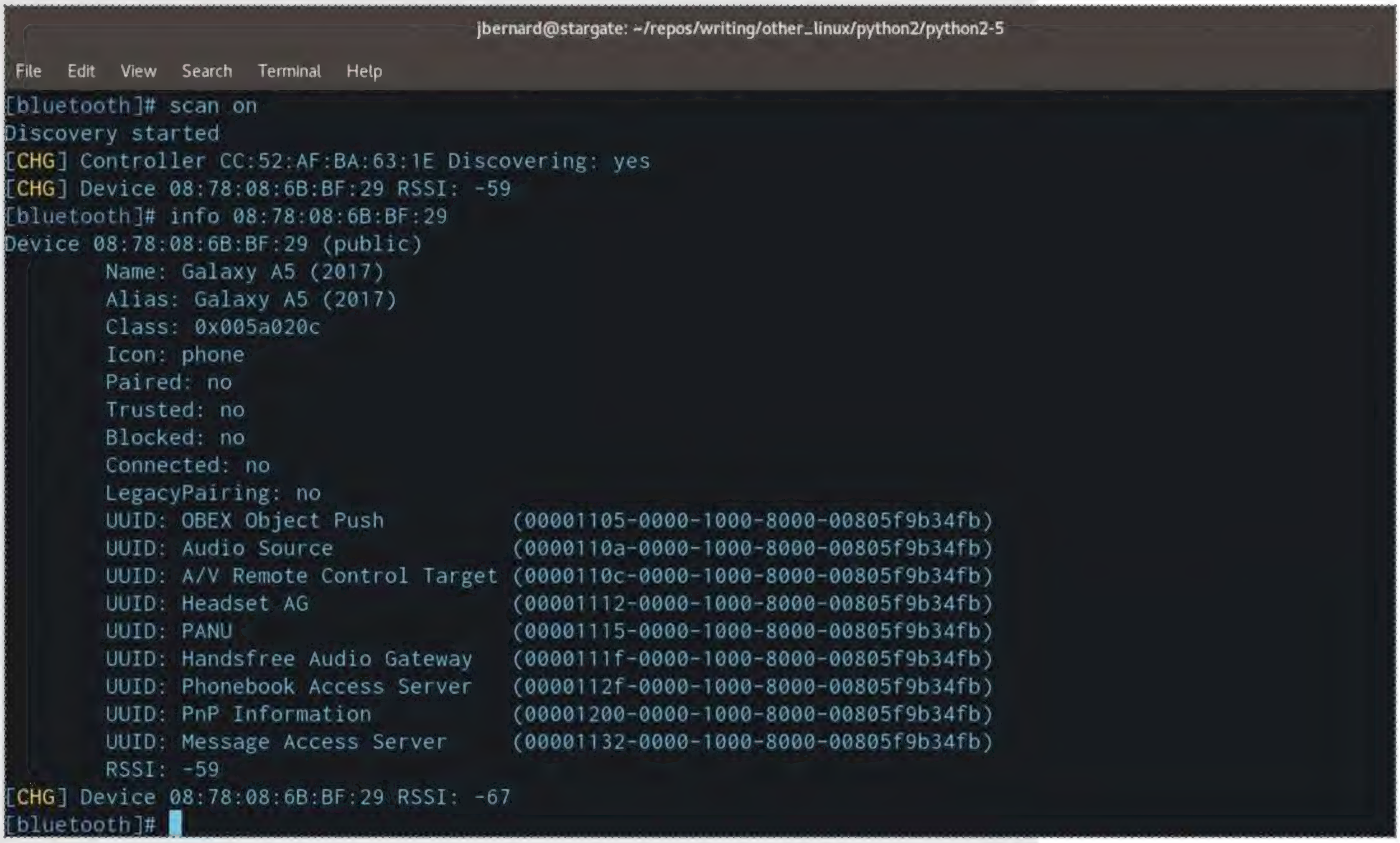
```
client_socket,address = server_socket.
  accept()
```

You can then call the recv(X) method of the client socket. The parameter X is the number of bytes to read in from this particular call. Once your code is all done with the Bluetooth connection, you need to remember to shut everything down cleanly.

```
client_socket.close()
server_socket.close()
```

Don’t forget that you have two socket connections to shut down. Now that the basics have been covered, we can dig into heavier functionality.

One of the things you may want to do with your Raspberry Pi is configure it to act as a data-collection point for whatever remote devices are communicating with it. In this case, you will want to dump the incoming data to a file for further processing later.



Above When using the command line utility bluetoothctl, you get to see all of the devices within range of your Raspberry Pi, including their hardware addresses

Copy files around

The first step is to accept an incoming connection. Since you want to accept a connection from any device out there, you need to listen on all ports.

```
server_sock=BluetoothSocket( RFCOMM )
server_sock.bind((“”,PORT_ANY))
server_sock.listen(1)
port = server_sock.getsockname()[1]
```

The last line gives you the port that is being listened on. The next step is to accept a connection and grab the device name from that incoming connection.

```
client_sock, client_info = server_sock.
  accept()
```

With this information, you can open up a unique file for this particular connection in order to save the incoming data to it. As a simple example, you could do something like the following.

```
client_file_name = client_info[0] + “.out”
client_file = open(client_file_name, ‘a’)
while True:
    data = client_sock.recv(1024)
    if len(data) == 0: break
    client_file.write(data)
```

The open() function uses the append mode, so that all of the data for a particular device gets collected in one location to make it easier to work with later on. Depending on what kind of data is coming in, you may want to do some formatting to make it easier to deal with.

Going in the other direction is not too difficult, either. You simply need to open a local file for reading, and then send the data out to the other device. A simple example would look like the following:

```
data_file = open(‘data.txt’)
while True:
    data = data_file.read(1024)
    if len(data) == 0: break
    client_sock.send(data)
```

This type of pattern should be familiar to anyone who has written TCP socket code before. With this boilerplate code, you could write a controller program for your Raspberry Pi that can send out data files to the remote

devices, and then read in results and save them to output files. These remote devices could be almost anything. You may have a collection of other Raspberry Pis, older smartphones that are lying around, or some custom hardware based on an Arduino with a Bluetooth module attached to it.

Create a server

If your Raspberry Pi is going to act as the central data-collection node, you'll want to actually create a server process that is able to listen for all incoming connections and process them as they report in.

The key is that you will need one thread of execution to handle incoming connections. This thread will add the connected sockets to a data structure. Then you will need another thread to go through these connections and deal with any incoming data. For both of these sections of code, we'll assume that you have a dictionary, named connections, that will be used to handle all of this processing.

The first step is to handle the incoming connections. For this portion, the code will essentially set up the socket and then sit and listen for the next connection request.

```
while True:
    socket, info = server_sock.accept()
    address, psm = info
    connections[address] = socket
```

This code snippet will loop forever, listening for incoming connections and saving them to the dictionary when they get established.

The next chunk of work is to check all of these

connections and see what might be coming in on them. The basic check would look like the following:

```
def check_data(address, output_file):
    socket = connections[address]
    while True:
        data = socket.recv(1024)
        if len(data) == 0: break
        output_file.write(data)
    socket.close()
    del connections[address]
```

Since the recv() function call is blocking, meaning that it will sit and wait until something happens, you will likely want to create a separate thread for each of these connections so that your program won't get blocked and frozen each time it checks for incoming data from each of these connections.

Also, the given function cleans up after itself when the connection is lost, or closed from the other side. This may be something that you would rather handle in some central fashion, depending on how robust you

want your program to be.

Control devices

When dealing with devices in your Bluetooth neighbourhood, there are a couple of other techniques that you may want to consider adding to your code. The first is code to advertise yourself to all of the locally visible devices, by using the Service Discovery Protocol (SDP). There is a function called advertise_service() that you can use for this. Assuming that you already opened up a socket that is currently listening, as we did earlier, the following code sets up the SDP process.

```
bluetooth.advertise_service(server_sock,
  “rpiService”, service_classes=[bluetooth.
  SERIAL_PORT_CLASS], profiles=[bluetooth.
  SERIAL_PORT_PROFILE])
```

This sets up a service, named rpiService, that other devices can look for and decide whether they should be connecting to or not. This can help make the setup of your remote devices a bit easier, since you are now dealing with names rather than addresses.

If your remote devices are also using Python code to connect (maybe they are also Raspberry Pi boards), the following code is an example of how to make that connection.

```
services = bluetooth.find_se
  rvice(name=”rpiService”,
  uuid=SERIAL_PORT_CLASS)
match = services[0]
port = match[“port”]
```

```
name=match["name"]
host=match["host"]
client_socket=BluetoothSocket( RFCOMM )
client_socket.connect((host, port))
```

This code assumes that there is only one service visible that has the name rpiService.

Along with devices being able to discover your Raspberry Pi server, you may need to enable your Raspberry Pi to discover what devices are visible to it. The function discover_devices() does this scan and reports back useful information about any that are found.

```
nearby_devices = bluetooth.discover_
 devices(lookup_names=True)
for address, name in nearby_devices:
    print("Found  %s - %s" % (addr, name))
```

This way, if your devices are configured to accept incoming connections, your Raspberry Pi can be completely in charge. It can look to see what devices are up and listening, and then go ahead and make socket connections to them and send them control signals. This way of working implies that the remote devices are coded by yourself and are operating on the premise that the Raspberry Pi will be connecting to them and controlling them directly.

Where next?

Before moving on, we need to stress that you should not use any of the code you have seen directly as written. As you should have noticed, we did no error-checking at all. With a protocol such as Bluetooth, which is designed

for short-range and low-power usage, connections between devices are a bit more fragile than you might be used to – especially if you have previously coded communication programs over a hard link, such as TCP over Ethernet.

This means that you should be responsible enough to write your code to be a bit more robust than my examples, especially if the remote devices are at the extremes of the radios' range. Having said that, we hope that this has put another option in your toolbox for when you need to have multiple machines talking to each other without needing to physically connect them.

Next issue

Get inspired · Expert advice · Easy-to-follow guides

"Create a retro-style Smartphone"

Get this issue's source code at:
www.linuxuser.co.uk/raspicode